Universidad Autónoma de Madrid
Escuela politécnica superior

Grado en Ingeniería Informática

Trabajo Fin de Grado

MODELOS DE REDES NEURONALES RECURRENTES EN CLASIFICACIÓN DE PATENTES

Autor: Guillermo Guridi Mateos
Tutor: Álvaro Barbero Jiménez
Ponente: José Ramón Dorronsoro Ibero

Junio 2017

MODELOS DE REDES NEURONALES RECURRENTES EN CLASIFICACIÓN DE PATENTES

Autor: Guillermo Guridi Mateos
Tutor: Álvaro Barbero Jiménez
Ponente: José Ramón Dorronsoro Ibero

Instituto de Ingeniería del Conocimiento
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Junio 2017

Resumen

Resumen

Este trabajo se centra en la clasificación automática de patentes usando redes LSTM (Long-Short Term Memory). Las patentes son documentos que recogen y acotan la propiedad intelectual, compuestas principalmente por texto y múltiples etiquetas correspondientes a una clasificación jerárquica de varios niveles (IPC).

Para empezar se describen distintos métodos de tratamiento y extracción de características de textos y de clasificación por aprendizaje automático. Éstos son las bolsas de palabras, los perceptrones, los perceptrones multicapa, los mapeos (embbedings), las redes neuronales recurrentes y por útlimo las redes recurrentes con neuronas LSTM. Además se revisan distintos artículos centrados en la clasificación de patentes para conocer el estado del arte y otras alternativas a los modelos que se van a probar.

A continuación se establece una metodología de desarrollo y pruebas de modelos para poder evaluar con facilidad los distintos experimentos. Dentro de esta metodología se incluye un *framework* de flujo de datos y entrenamiento de modelos así como pequeñas utilidades para poder replicar todos los experimentos en entornos virtualizados con *docker* pero con acceso a GPUs. Además, este *framework* realiza un guardado automático de los resultados intermedios de los distintos módulos de procesado de texto (que también ha habido que implementar) para así conseguir que operaciones costosas previas al entrenamiento se ejecuten solo cuando sea necesario y no más de una vez.

Finalmente se realzó un análisis estadístico de los textos que se iban a usar para entrenar los modelos para poder decidir razonadamente que tratamiento usar para los modelos definitivos. También se describe la métrica que se va a usar para evaluar los distintos modelos así como ciertos parámetros de entrenamiento. Por último se realizaron unos experimentos con distintas variaciones de modelos con LSTM, algunos con un mapeo precalculado y otros entrenando un mapeo de cero. La gran mayoría fueron poco conclusivos y no tuvieron resultados muy buenos, pero dejan abierta la puerta a sencillas mejoras e indican los pasos a seguir para mejorar la precisión.

Palabras Clave

IPC, patente, LSTM, clasificación de textos, NLP, aprendizaje automático, redes neuronales

Abstract

The main topic of this project is automatic patent classification using LSTM networks (Long-Short Term Memory). Patents are official documents which describe and limit intelectual property for ideas. It's content is mainly text, and multiple labels from a hierarchical multi-level classification system (IPC).

It begins with the description of several methods for treatment and extraction of text features as well as machine learning classification. The ones described are bag of words, perceptrons, multi-layer perceptrons, embbedings, recurrent neural networks and, at last, LSTM networks. To get an overview of the state of the art in patent classification and to take a peek at different methods than those described before, several papers which show promising results are summarized.

Afterwars, a testing methodology to be able to test different models with ease is stablished. This methodology encloses a newly developed framework that allows for easy building of data workflows and provides some useful utilities to to guarantee reproducibility of the experiments, specially inside docker containers with their own GPU. Furthermore, this framework does some automatic caching to prevent big calculations that have alredy been done from happening twice.

Lastly, an estatistical analysis is performed on the texts meant to be used as training data. This will give us some insight that will help us choose the definitive model and its etl. pipeline. A performance metric is also defined to be able to have some criteria over which models and parameters work better and we shoud spend more time on. Finally, some experiments were attempted, all using LSTM networks. The results don't come out quite good but they leave plenty of room open for improvement as well as guidelines marked by those tests.

Key words

IPC, patent, LSTM, text classification, NLP, machine learning, neural networks

Índice general

Índice de Figuras

Índice de Tablas

Glosario

backpropagation Algoritmo de propagación de gradientes a través de una red neuronal, para su correcta actualización durante la fase de entrenamiento.. 7

Descenso por gradiente Algoritmo que en base a un gradiente decide cuánto se actualizan los pesos, en redes neuronales se usa junto con backpropagation.. 7

docker Sistema de virtualización muy liviano basado en contenedores.. 14

dropout Probabilidad asignada a capas de redes neuronales, que indica como de probable es que esa capa no lea alguna de sus entradas.. 11

función escalón de Heaviside Función umbralizadora que devuelve 0 para cualquier entrada por debajo del umbral e 1 para las demás.. 6

lstm Tipo de neurona que está conectada consigo misma y tiene puertas que le permiten mantener información guardada un número indefinido de pasos.. 5

mapeo Función de un espacio mayor, como puede ser un diccionario de palabras a un espacio menor, procurando conservar la mayor parte del significado que sea posible.. 8

one hot encoder Codificación binaria para variables categóricas en la que cada categoría corresponde a un vector de 0s excepto por una columna (o fila) que es 1.. 6

one-vs-all Método para convertir un clasificador mono-etiqueta en uno multi-etiqueta, combinando n de los primeros.. 6

perceptrón Neurona básica de una red neuronal compuesta de un vector de pesos, un sesgo y una función regularizadora.. 6

word2vec Mapeo de palabras a vectores dado por un diccionario que se puede entrenar con distintos problemas de clasificación.. 8

época Étapa de entrenamiento de algoritmos supervisados en la que se revisan todos los datos del dataset (antes de volver a empezar con la siguiente).. 12

Glosario de acrónimos

- **BPTT**: BackProgation Through Time
- **GPU**: Graphics Processor Unit
- **IIC**: Instituto de Ingeniería del Conocimiento
- **IPC**: International Patent Classification
- **LSTM**: Long-Short Term Memory
- **RNN**: Recurrent Neural Network
- **SESIAD**: Secretaría De Estado para la Sociedad de la Información y la Agenda Digital
- **SVM**: Support Vector Machine
- **WIPO**: World Intellectual Property Organization

1 Introducción

1.1. Motivación

Las patentes son documentos oficiales en los que se reconoce la propiedad intelectual de una invención. Éstas se clasifican según un sistema de clasificación internacional (IPC) regulado por la Organización Mundial de la Propiedad Intelectual (WIPO). Cada patente puede obtener una o varias etiquetas de clasificación.

Las categorías de las patentes están organizadas jerarquicamente, con 8 secciones en el nivel más alto, y cada vez más subdivisiones hasta unas 70000 clasificaciones distintas en el nivel más bajo.

La clasificación es útil para muchos individuos, sobre todo duránte una búsqueda del "estado de la técnica". Esta búsqueda la realizan las autoridades que conceden patentes, las unidades de investigación y desarrollo, los eventuales interventores, o cualquiera interesado en los últimos avances de una tecnología. Por ésto, obtener un método que ayude en la clasificación de manera pasiva (sugiriendo etiquetas pero no decidiendo resultados finales), puede mejorar mucho el proceso actual de clasificación.

El Instituto de Ingeniería del Conocimiento (IIC) ya ha desarrollado tal sistema, entregado a la Secretaría de Estado de la Sociedad De La Información Y Agenda Digital (SESIAD) para ayudar con la clasificación (y búsqueda de similitudes) de las, aproximadamente, 3000 solicitudes de patentes que reciben cada año. En este trabajo se examinarán más ténicas de análisis y clasificación de textos y se intentará probar (usando redes neuronales recurrentes) distintos modelos a los usados en el sistema entregado para evaluar su eficacia.

1.2. Alcance

En este trabajo se intentará determinar las distintas etiquetas de clasificación de una patente en función de su contenido (título, descripción y revindicaciones).

Para ello haremos un repaso de los métodos principales de clasificación automática para entender lo que nos aportan las redes neuronales recurrentes y cómo debemos intentar usarlas. Además veremos algunas publicaciones con métodos diferentes que consiguen buenos resultados.

A pesar de todas las categorías existentes en el IPC nos centraremos en el nivel más alto de la jerarquía por el tiempo y los recursos necesarios para crear modelos para todas las subdivisiones de los modelos posteriores. Sin embargo, las pruebas con este nivel servirán para evaluar cómo de buenos son los modelos creados para su posterior ajuste y entrenamiento con los niveles más profundos de la clasificación.

La pieza fundamental que se usará en los modelos evaluados son las redes neuronales recurrentes (RNN), pero dada la genericidad de éstas, este trabajo se centrará en intentar replicar algunos modelos que hayan sido probados con anterioridad en problemas parecidos.

Ya que las redes neuronales necesitan que los datos de entrada sean vectoriales, probaremos distintas maneras de representar el texto de entrada, como *embeddings* (aprendidos o pre-generados) de cada palabra.

1.3. Objetivos

Los objetivos de este trabajo son los siguientes:

- Revisar el estado del arte en redes recurrentes y otros modelos para la clasificación automática de textos.
- Investigar qué modelos se han aplicado a este problema en concreto y los resultados que se han obtenido.
- Diseñar e implementar herramientas básicas para facilitar la prueba y verificación de distintos modelos de clasificación.
- Implementar y probar distintos modelos de aprendizaje automático que usen redes neuronales recurrentes
- Programar el procesamiento y limpieza de datos necesario para que los modelos implementados puedan recibir el texto y las clasificaciones adecuadamente desde los archivos de patentes.
- Evaluar los distintos modelos generados y compararlos con los resultados anteriores.

1.4. Estructura del documento

El primer capítulo de este documento realiza una leve introducción al problema a tratar y las característas más relevantes de éste como problema de aprendizaje automático, poniendo en contexto la importancia de su realización.

En el segundo capítulo se comentan y explican algunas de las ténicas más usadas para la clasificación de textos y resultados recientes en clasificación de patentes. Además se detallarán las tecnologías usadas en el desarrollo de este trabajo.

El tercer capítulo detalla el proceso seguido para la implementación de los modelos probados y cómo se realizaron todos los entrenamientos y las pruebas. También habla sobre la elección de los modelos probados.

En el cuarto capítulo se muestran los resultados obtenidos por los distintos modelos probados y ciertas métricas obtenidas sobre los datos originales.

Por último, el quinto capítulo cierra el trabajo con algunas conclusiones y lineas de trabajo futuro.

2

Estado del Arte

La clasificación de textos, junto con muchos otros problemas abarcables por el *Machine Learning*, es un problema abierto con bastante popularidad actualmente por lo que el estado del arte avanza a una velocidad vertiginosa. Nuevas técnicas planteadas para otros problemas son rápidamente evaluadas para éste, haciéndo dificil mantenerse al día y probar las técnicas más avanzadas. Incluso si acotamos nuestra búsqueda a soluciones basadas en redes neuronales recurrentes, la variabilidad dentro de los modelos y el procesamiento de datos es muy alta.

2.1. Clasificación de textos

El problema a tratar en este trabajo se engloba dentro de una categoría más general, que es la clasificación de textos. En este proceso de clasificación o etiquetado se pueden asignar varias etiquetas a un solo texto. Gran parte de la literatura al respecto habla de etiquetar solo con una etiqueta cada texto, pero muchos de los modelos usados son generalizables a varias etiquetas.

A continuación vamos a ver algunos de los conceptos aplicados a la clasificación de textos que más relevancia tienen respecto a los métodos planteados para este trabajo. Existen otros métodos, algunos muy recientes, que no van a ser detallados aquí pero que podrían ser muy buenos candidatos para extender el trabajo iniciado con los modelos seleccionados.

Para desarrollar esta sección vamos a empezar por modelos bastante sencillos e incrementalmente describir modelos cada vez más complejos hasta llegar a las redes *lstm* para, de esta manera, entender la evolución que ha tenido el problema de las clasificación de textos.

2.1.1. Modelo bolsa de palabras

Este modelo se usa para representar documentos en base a las palabras que contienen. No tiene en cuenta el orden en el que aparecen las palabras pero sí la frecuencia con la que aparecen. Para entrenar un modelo de bolsa de palabras se toman todas las palabras distintas del corpus (o un subconjunto en concreto) y se le asigna a cada una de ellas un índice único. A continuación se lee cada texto y se genera un vector de frecuencias. En este vector, el elemento i será igual al número de apariciones de la palabra correspondiente al índice i en el texto, siendo 0 si no está en el texto.

Teniendo estos vectores no queda más que aplicar un modelo lineal a la clasificación, como podría ser una regresión logística, que tomará las frecuencias de las palabras (en forma de vector) como entrada y su salida será la etiqueta del texto. Para casos multieqtiqueta se puede usar una estrategía de *one-vs-all*. En ésta se entrenan N modelos binarios, uno para cada clase posible, que intentan predecir la pertenencia o no a esa clase independientemente de las demás.

2.1.2. Perceptrón

Un perceptrón es la unidad básica normalmente usada en las redes neuronales. Por sí solo es un modelo lineal que puede tomar como entrada un número arbitrario de variables y obtener como resultado una respuesta binaria. Se define por un vector de pesos $\bar{w}$ que se aplican a cada una de las entradas x_i, su sesgo y la función de activación que se coloca a la salida. Para entender mejor como funciona, veamos la fórmula que define la salida de un perceptrón:

$$f(\bar{x}) = \begin{cases} 1 & \text{si } \bar{w} \cdot \bar{x} + b > 0 \\ 0 & \text{en otro caso} \end{cases} \tag{2.1}$$

siendo $\bar{w} \cdot \bar{x} = \sum_{i=1}^{N} w_i x_i$ ($\bar{x}$ es un vector de elementos x_i)

Lo que esto nos indica es que la salida depende de la suma de los valores de entrada (multiplicando cada uno de ellos por su peso w_i) y sumándo al final un sesgo fijo. Si este valor es mayor que 0, entonces la salida de la neurona es 1, si no 0. Este tipo de función de activación se llama la función escalón de Heaviside.

Éste modelo, en base a características del texto como el vector de la bolsa de palabras descrito anteriormente puede clasificar textos para una categoría. Ampliando el modelo con perceptrones independientes para cada categoría se tendría un modelo de clasificación como el que buscamos, válido. Pero estos perceptrones tienen una limitación, solo pueden separar linealmente, así que solo serán capaces de clasificar correctamente nuestros textos si en el espacio vectorial de nuestros vectores caracterizadores, las categorías son separables del resto por hiperplanos. La mayoría de las veces, especialmente con texto, esto no se cumple, así que debemos buscar modelos que puedan capturar una mayor complejidad.

2.1.3. Perceptrón multicapa

Estas son las redes neuronales clásicas, compuestas de perceptrones conectados de manera que no formen un ciclo. En este tipo de redes la información se desplaza solo hacia delante hasta la salida de la red. Estas redes observan todo un ejemplo entero en la entrada y generan una sola predicción.

Construir un sistema de clasificación de textos usando redes neuronales implica generar una representación numérica de tamaño fijo de cada texto a clasificar. Como en los casos anteriores la entrada debe ser un vector. Aparte del vector bolsa de palabras comentado anteriormente se pueden tomar otras características del texto, o incluso un vector combinación de vectores obtenidos de las palabras del texto o alguna otra aproximación similar que consiga caracterizar bien los textos (como se puede observar en la figura 2.1).

Así mismo se debe codificar también la salida que queremos que aprenda la red. Se puede simplemente asignar un número a cada categoría, pero en este tipo de modelos es más común codificar las clasificaciones con un sistema conocido como *one hot encoder*. En éste, a cada categoría se le asigna un número, y después un vector de dimensión N de ceros excepto por un uno en el número de la categoría representada, siendo N el número de clases posibles en las que

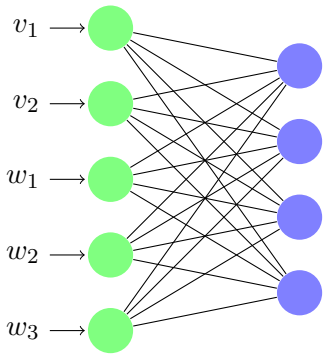

Figura 2.1: Combinación de vectores a la entrada de una red neuronal

se quieren clasificar. Para multiples etiquetas se pueden poner a uno varias de las dimensiones del vector (figura 2.2).

clasificaciones	Vector							
H	0	0	0	0	0	0	0	1
H y G	0	0	0	0	0	0	1	1
B y G	0	1	0	0	0	0	1	0

Figura 2.2: Conversión de categorías a *k-hot-encoded*

Las redes neuronales se componen de capas de perceptrones conectadas en serie de manera que cada capa use los valores de la capa anterior y la salida de los últimos perceptrones se tome como resultado. Como ya hemos visto, cada perceptron está compuesto de ciertos parámetros, que son los pesos que da a los valores de entrada, un sesgo y una función de activación que regulariza la salida. Estos pesos se pueden actualizar con un algoritmo conocido como *backpropagation*, en donde se alteran en la dirección contraria al gradiente de la función objetivo para conseguir que su predicción se aproxime cada vez más a la salida esperada.

Este algoritmo se conoce como Descenso por gradiente y funciona de la siguiente manera:

Tomemos un solo perceptrón de la última capa de la red. Cuando se realiza una predicción, éste emite un resultado $\hat{y} = f(\bar{w} \cdot \bar{x} + b)$. El resultado se compara con y mediante una función de pérdida (examinaremos la usada en nuestro modelo más adelante) que se quiere minimizar. Para saber como actualizar los pesos tomaremos $\frac{\partial E(y,\hat{y})}{\partial w_i}$, es decir, la derivada de la función de error respecto de cada uno de los pesos. Actualizaremos cada peso sumándole la derivada multiplicada por un factor (conocido como factor de aprendizaje) que regula la velocidad del descenso:

$$W_i^{t+1} = W_i^t - \gamma \frac{\partial E(y, \hat{y})}{\partial w_i^t} \tag{2.2}$$

El mismo tipo de ecuación se puede obtener para calcular el nuevo sesgo a cada paso. Y está muy claro para una sola capa, pero cuando tenemos varias capas en una red (figura 2.3), debemos usar la regla de la cadena para poder derivar la función de pérdida de neuronas internas.

Para el caso de la clasificación de textos en varias clases se colocan n perceptrones a la salida. Si se trata de un problema con una sola etiqueta, se cogerá el valor más alto entre los n posibles a la salida del modelo. Si estamos tratando con un problema multietiqueta, como es el caso de las patentes (pueden pertenecer a varias categorías), se aplicará una función sigmoidal[1] a la salida de cada perceptrón para normalizar sus valores al intervalo $[0, 1]$. Estos valores se pueden

[1]La función sigmoidal transforma valores de todos los reales a valores en el intervalo $[0, 1]$ conservando el 0 como punto fijo. Queda definida por $\sigma(x) = 1/(1 + e^{-x})$

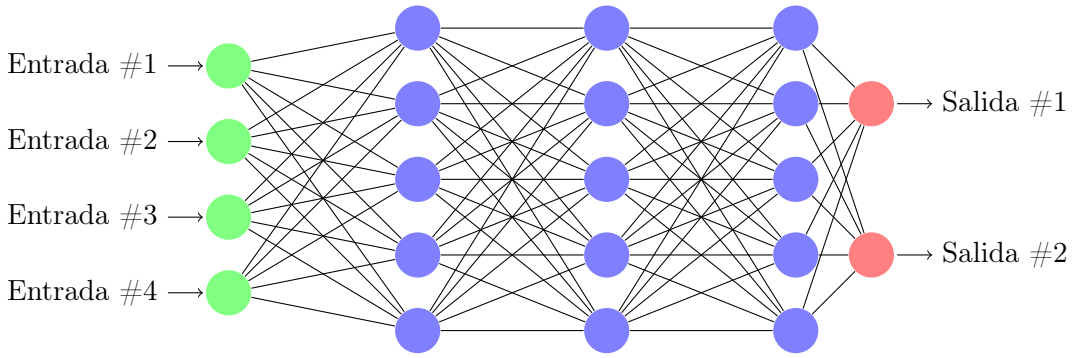

Figura 2.3: Red neuronal profunda

interpretar como la probabilidad de que un texto pertenezca a la clase i (i siendo la posición de ese perceptrón en la última capa). Por último, para escoger las etiquetas se aplicará algún tipo de función umbralizadora a la salida (típicamente $\hat{y} > 0{,}5$) para después tomar las salidas que cumplan la condición.

2.1.4. Mapeos (embeddings)

Matemáticamente se definen los mapeos como representaciones de objetos de un espacio en otros de menor dimensión. En el caso de el análisis del lenguaje natural se suele denominar "Word Embbeding" a las transformaciones de palabras de nuestro lenguaje (una dimensión por cada palabra) a vectores de un espacio continuo de dimensión N donde, con suerte, las dimensiones cobren cierto significado semántico.

Es importante estudiar esta técnica cuando se habla de procesamiento de texto ya que es una de las mejores maneras que tenemos de convertir palabras a vectores que puedan entender nuestros modelos. Google ha conseguido resultados muy buenos con su sistema *word2vec* con el que muestra relaciones semánticas entre palabras o parejas de palabras que se ven reflejadas en su diferencia de vectores.

Para entrenar un modelo como *word2vec* se establece un diccionario de conversión de palabra a vector inicializado aleatoriamente y se entrena actualizando cada vector poco a poco con muchos ejemplos. El problema es que no tenemos claro cuál es el objetivo con el que tenemos que entrenar para decidir como actualizar el diccionario, ya que no disponemos de información de la semántica de cada palabra. Por tanto, usaremos otra tarea de aprendizaje automático para entrenar este diccionario. En el caso de *word2vec*, tomaron una multitud de fragmentos de frases con una longitud en palabras fija, junto con versiones alteradas en las que la palabra central era sustituida por una palabra aleatoria. Por último conectaban el diccionario de codificación a una red neuronal que intentaba decidir si la frase de entrada correspondía al corpus original o al alterado. Al entrenar ambas partes juntas, se "fuerza" al diccionario a generar vectores con significado semántico para que la red sea capaz de discriminar correctamente las frases.

Para este trabajo utilizaremos dos aproximaciones. La primera será replicar este modelo de entrenamiento desde cero, pero tomando como tarea adicional para generar vectores la propia clasificación de patentes. Por otro lado tomaremos los vectores pregenerados por *word2vec* que ya incluyen cierta semántica general y los usaremos como datos de entrada de manera que solo tengamos que entrenar la red neuronal.

2.1.5. Redes neuronales recurrentes

Las redes neuronales recurrentes son una evolución de los perceptrones en las que puede haber ciclos en las conexiones entre las neuronas de cada capa. Ésto las convierte en buenas candidatas para trabajar con datos con dependencias estructurales en una dimensión, como el texto, o el audio. A veces se usan solas, pero lo más normal es que sirvan para generar características útiles para una red neuronal de perceptrones que realiza el último trabajo de relación y predicción. Ocasionalmente se pueden usar para generar secuencias, aunque ese no será su uso en este trabajo.

El que existan ciclos en las conexiones de la red neuronal permite librarnos de la relación "un patrón de entrada ⇒ un patrón de salida". Ahora hay algunas neuronas que tienen a la vez el vector procedente del patrón actual a la entrada de la red y vectores que generaron ellas mismas en iteraciones anteriores. Estas redes van a tener que decidir en cada iteración cuánto peso le dan a la información que ya tenían y cuánto a la información nueva. Si una neurona decide darle mucha importancia a su valor acumulado durante muchos pasos, ésta conseguirá que se puedan relacionar datos obtenidos en elementos de la secuencia muy separados.

Una manera bastante clara de entenderlas es desenrollarlas en el tiempo y observar su similitud con las redes neuronales tradicionales. Este desenrollado también nos ayuda con su entrenamiento, que se puede realizar con *BPTT (BackPropagation Through Time)*, una variante del *backpropagation* que ya hemos visto. Pero este tipo de entrenamientos sufre un problema conocido como *Vanishing gradient*. Cuando desenrollamos el bucle temporal de la neurona obtenemos una cadena muy larga desde la entrada a la salida. Si intentamos actualizar los pesos como se contaba antes, usando la regla de la cadena, nos daremos cuenta de que el gradiente aplicado a las neuronas más próximas a la salida es mucho mayor que el que se aplica a las neuronas más próximas a la entrada. En otras palabras, las neuronas de capas finales aprenden más rápido que las de las primeras capas. Esto es un problema ya que sin valores adecuados en las primeras capas no se pueden capturar los problemas complejos para los que estas redes fueron concebidas.

Se han desarrollado muchas técnicas para paliar este efecto, como funciones de regularización especiales, o métodos de inicialización de pesos que tienen en cuenta este efecto. Sin embargo, hay alternativas mejores.

2.1.6. Redes LSTM

En teoría las redes neuronales recurrentes estándar pueden modelar cualquier tipo de relación entre partes de la secuencia, pero cuanto más grande se vuelve ésta, más dificil lo tienen. Las neuronas LSTM (Long Short Term Memory) surgen para arreglar este problema. Son un bloque de construcción que se puede usar repetidas veces y que permite la asignación y lectura "automática" de valores por parte de la red. Para entenderlas mejor, veamos su estructura[2]:

[2]Hay muchas variaciones de las neuronas LSTM, pero aquí se muestra una de las más generales

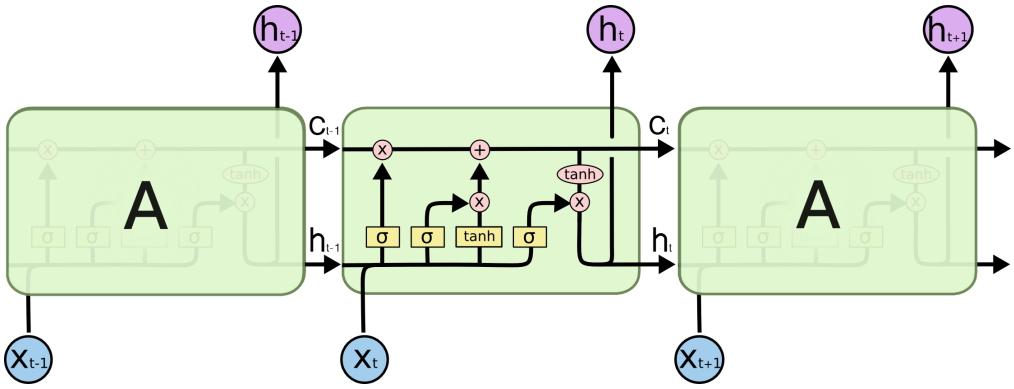

Figura 2.4: Neurona lstm
Esquema realizado por Christopher Olah. http://colah.github.io

Cada linea mostrada en la figura 2.4 no es sólo un número como en el caso de los perceptrones, sino que representa un vector entero de la dimensión de los elementos de la secuencia de entrada.

Como podemos ver, la neurona genera dos salidas, h_t y C_t. h_t es la salida de la neurona que se propaga a las siguientes capas de la red y C_t es el estado interno de la neurona. Ambos se propagan también a la neurona en el siguiente estado de tiempo, convirtiéndose en h_{t-1} y C_{t-1}. Además tenemos x_t como entrada, que representa el elemento de la secuencia siendo leído a cada paso.

Existen tres puertas lógicas dentro de la neurona. Cada una de estas puertas lógicas está conectada con los resultados del paso anterior h_{t-1} y el nuevo elemento de la secuencia x_t, y usa éstos, actuando como un perceptron con función de activación entre 0 y 1, para cambiar el comportamiento de la neurona. La primera puerta es la puerta que decide cuánto se recuerda del estado anterior C_{t-1}, multiplicando la salida de la puerta por h_{t-1}. Después se añade al valor obtenido una multiplicación del nuevo valor x_t por el resultado de la segunda puerta lógica, la puerta de actualización. El resultado de esa operación se convierte C_t. Por último h_t toma el valor de C_t regularizado y multiplicado por la última de las puertas, la de salida. En conjunto, esta sería una aproximación de alto nivel a las funciones de actualización de una neurona LSTM:

$$C_t = C_{t-1} \cdot \text{puerta1}(h_{t_1}, x_t) + \text{puerta2}(h_{t_1}, x_t) \cdot W(h_{t_1}, x_t)$$
$$h_t = C_t \cdot \text{puerta3}(h_{t_1}, x_t)$$

En esta arquitectura vemos que la función de activación de estas neuronas es la función identidad ($f(x) = x$), junto con el peso de la puerta de recordar. Esta será muy próxima a 1 si se quiere recordar y evitará que el gradiente se disipe. Además, como no puede ser mayor que 1, también evitará que el gradiente explote. Esto las convierte en buenas candidatas para el análisis de secuencias.

Se espera que estas redes ayuden en la clasificación de texto al ir reteniendo características encontradas en diversas partes del texto generando un set de características útil que pueda recibir la red clasificadora.

Variaciones de neuronas recurrentes

Existen muchas variantes de las neuronas recurrentes y, en concreto, de las redes LSTM que hemos descrito. Por ejemplo, las fórmulas mostradas no tienen en cuenta el valor de C_{t-1} para tomar la decisión de las puertas. Si se incluye esta conexión se dice que la neurona LSTM puede

hacer *peeping* (asomarse) a su estado. También existe una variante de gran popularidad por su simplicidad en la que se juntan las puertas de olvido y de entrada en una sola.

2.1.7. Hiperparámetros de entrenamiento

Hablaremos más en detalle de las decisiones que se han tomado en este trabajo sobre los parámetros de entrenamiento en los capítulos Diseño y desarrollo y Pruebas y Resultados, pero aquí introduciremos el marco teórico de estos conceptos.

Topología de las redes

La forma (número de capas, nodos por capa, disposición) de las redes neuronales no es un problema trivial. Este es uno de los hiperparámetros más dificil de elegir y ajustar y con el que más nos tendremos que guiar de experimentos anteriores. Tiene mucho efecto en la cantidad de memoria requerida para un algoritmo y su tiempo de entrenamiento. Por supuesto, también en su capacidad de predicción. En general modelos con más capas o neuronas serán capaces de predecir relaciones más complejas pero también son más propensos al sobreajuste. Por otro lado, un modelo con pocas neuronas es poco probable que sobreajuste, pero a lo mejor no es capaz de capturar los patrones subyacentes del problema.

Dropout

Las redes neuronales con muchas conexiones, como las redes neuronales profundas o los modelos recurrentes como el que hemos visto, tienen una gran cantidad de parámetros (o pesos). Ésto hace que a la red le sea bastante sencillo sobreajustar los datos. Una buena solución a este problema es usar *dropout*, que significa que con cada lote (definido a continuación), en vez de tener en cuenta la red entera, vamos a escoger una fracción de éstos y vamos a tomarlos como una red completamente independiente. Con el siguiente lote de datos se repetirá lo mismo con otros nodos escogidos aleatoriamente. Esto evita que se creen conexiones con pesos muy específicos para ejemplos concretos del set de datos. Como parámetro de *dropout* se puede escoger cualquier número entre 0 y 1, significando el 0 que siempre se coge la red entera y el 1 que siempre se coge una red sin nodos.

Tamaño del vocabulario

Cuando se trabaja con mapeos de palabras hay que elegir un tamaño fijo para la tabla de palabras que se entrena junto con las redes neuronales. El reducir esta variable hace que se pierdan palabras que aparezcan menos frecuentemente en el texto, ya que son de las que menos se puede generalizar. Considerar muchas palabras, en general es una buena idea (aunque puede llevar a sobreajuste) a excepción del coste computacional añadido. Como veremos, el tamaño de vocabulario también se puede reducir más si quitamos las palabras vacías (conocidas como *stop-words*) sin significado semántico evidente.

Tamaño de secuencia

Aunque potencialmente pueden absorber entradas de cualquier longitud, el entrenamiento de las redes recurrentes funciona mejor si se puede desenrollar el lazo que las une a sí mismas tanto como haga falta hasta que se convierta en un perceptrón multicapa. Como las secuencias son de longitud variable, las más largas se van a a truncar al tamaño establecido, mientras que

las que sean más cortas que ese tamaño serán ampliadas por un *padding* que después es ignorado a la entrada de la red. Cuanto mayor hagamos el tamaño de secuencia, más información seremos capaces de obtener de el texto original pero más costoso será el entrenamiento y puede llegar a no converger correctamente si es demasiado larga. Por supuesto, también conlleva un uso de memoria a tener en cuenta.

Tamaño de lotes

Los lotes son los bloques de datos que se usan en una red neuronal para ejecutar una actualización de gradiente. Es decir, para realizar una actualización, se coge un número de ejemplos del set de entrenamiento equivalente al tamaño de lote. Todos esos ejemplos son propagados por el estado actual de la red. Los resultados obtenidos se comparan con los resultados reales y se calcula la media de la desviación del resultado respecto a la clasificación real. Este valor se propaga hacia atrás por la red para actualizar los pesos.

Cuanto más grande sea el lote obtenemos más rendimiento ya que se pueden pasar muchos ejemplos de golpe por la red. Ésto nos da una función de error con menos varianza, que converge más rápido (a costa de analizar más ejemplos) ya que se pueden tomar pasos más grandes hacia el mínimo, claro que se realizan menos actualizaciones por época. Se debe intentar, para cada problema, encontrar un punto medio entre lotes muy pequeños y lotes muy grandes que funcione bien.

Épocas

Las épocas son el número de veces que la red neuronal va a repasarse todos los datos. Cuando se acaba una, vuelve a empezar otra. Con cada época mejora la precisión del modelo, al actualizar más veces sus pesos, pero también aumentan las posibilidades de sobreajustar los datos de entrenamiento. Por ello se suele usar una técnica llamada *Early Stopping*, por la cual el entrenamiento sigue de manera indfinida hasta que el acierto de un conjunto de datos independiente del de entrenamiendo no mejora en un número preestablecido de épocas.

2.2. Clasificación de patentes

Se han encontrado varias publicaciones centradas específicamente en la clasificación de patentes, de varios años y, a continuación, se van a analizar sus resultados.

2003: C. J. Fall et al.: Computer-Assisted Categorization of Patent Documents in the International Patent Classification

En esta publicación analizan patentes en cuatro lenguajes e intentan establecer un modelo común para ellas. Usan un modelo similar a la bolsa de palabras con un algoritmo ventana y una red neuronal para capturar correctamente la distribución de frecuencia de las palabras.

Entrenaron sus modelos de manera jerárquica para los primeros tres niveles de la clasificación, cada uno con precisión decreciente. Además, entre los lenguajes (inglés, francés, ruso y alemán) el mejor clasificado es el inglés.

2011: Ian Christopher et al.: Automated Patent Classification

Estos investigadores cogieron el set de patentes en inglés y realizaron el siguiente tratamiento de datos: Primero *tokenizaron* el texto dividiéndolo en palabras ignorando los símbolos de puntuación. Después realizaron *stemming*, que consiste en reducir las palabras a sus raices semánticas. A continuación construyeron una bolsa de palabras que finalmente fue reducida en dimensionalidad por medio de un análisis de semántica latente.

Este análisis se basa en tomar la matriz compuesta por todos los vectores de la bolsa de palabras (una patente por columna y una palabra por fila) y reducir las dimensiones de ésta. Para ello se toman palabras que tengan una distribución parecida y se intenta crear una sola fila para ellas cuyo valor sea una combinación lineal de los valores independientes de cada una de las filas eliminadas. Intentando que, a pesar de quitar filas, se pierda la menor cantidad de información posible.

Para el modelo probaron tres alternativas: regresión logística, una SVM (Support Vector Machine) lineal y un perceptrón multicapa. Durante su investigación se centraron en clasificar, principalmente, la primera capa de las etiquetas, aunque hicieron algunos experimentos con la segunda. Sus resultados son bastante prometedores, consiguiendo una medida-F (que definiremos más adelante) en torno al 94 %.

2015: Dilesha Seneviratne et al.: A Signature Approach to Patent Classification

Los modelos descritos en esta publicación fueron probados con varias partes de los textos de las patentes: el título, el resumen, las reivindicaciones y las primeras 300 palabras de la parente. El método que usan consiste en construir un vector para cada patente mediante la suma de vectores generados pseudo-aleatoriamente para cada palabra. Después usan una función de *hashing* sensible localmente para convertir ese vector a una firma de la patente. Para predecir, después de haber procesado la patente a evaluar, se buscan sus k vecinos más cercanos y se escoge la clase mayoritaria entre estos. Las subsecuentes categorías se obtienen escogiendo el subconjunto de vecinos que "ganó" la votación anterior y volviendo a votar en la segunda categoría. Como función de distancia usan la de Hamming (número de ediciones necesarias).

Sus resultados son bastante prometedores sobre metodos anteriores, aunque sus valores de precisión no muy sorprendentes al referirse a predecir correctamente 4 niveles enteros.

Instituto de Ingeniería del Conocimiento

En el proyecto realizado por el Instituto de Ingeniería del Conocimiento se evaluaron patentes en inglés y en español. El método usado para la clasificación fueron redes de perceptrones multicapa con una representación vectorial del texto como entrada. La manera de crear estos vectores fue distinta por cada lenguaje para ofrecer la mejor precisión.

Los modelos que programaron son jerárquicos, es decir, se ha entrenado uno para el primer nivel de la clasificación, luego otro modelo por cada categoría para clasificar en subcategorías, y así sucesivamente hasta el tercer nivel.

Los resultados fueron satisfactorios y acordes con lo encontrado en algunos de los artículos que hemos visto.

2.3. Tecnologías a utilizar

Para implementar este tipo de modelos que hemos descrito y probar su efectividad se ha optado por usar *python* y diversas librerías útiles para el aprendizaje automático y la ciencia de datos en general. Podemos dividirlas en tecnologías usadas para el procesado de datos, para los modelos, y tecnologías de apoyo, para la ejecución de todo esto y la construcción de los informes de resultados.

2.3.1. Limpieza y tratamiento de datos

Para esta tarea se han empleado módulos bastante comunes de *python* en estos ámbitos como:

- *lxml*: Módulo de python que provee funciones de parseado de *xml*, las cuales fueron muy útiles para leer los archivos de patentes y extraer la información requerida.
- *numpy*: Librería que implementa matrices y vectores de escalares y reales, junto con operaciones eficientes sobre ellas.
- *pandas*: Librería que trae el paradigma de tratamiento de datos más común de R, los *dataframes*, a python. Básicamente se trata de una librería que permite tratar con grandes cantidades de datos como si fueran tablas.
- *scikit-learn*: Librería que implmenta muchas operaciones comunes últiles para proyectos de aprendizaje automático, como la división de los datos en sets de entrenamiento, test y validación; o el cálculo de las métricas de evaluación de los modelos.
- *keras*: Aunque no sea su próposito directo, esta librería centrada en redes neuronales provee algunas funciones importantes para el procesado de secuencias de texto para posteriormente introducirlo a un modelo.
- *nltk*: Usada para obtener algunos recursos lingüisticos como las palabras vacías (*stopwords*).

2.3.2. Modelos

Los modelos han sido elaborados con ayuda de la librería mencionada anteriormente *keras*. Ésta es una interfaz de alto nivel para la definición de redes neuronales de muchos tipos. Por debajo puede trabajar con distintos *backends* de optimización: *theano* y *tensorflow*. En este trabajo se ha utilizado *tensorflow* por integrarse más facilmente con las GPUs del sistema de pruebas.

2.3.3. Apoyo

Para aislar la ejecución de estas pruebas de otros usuarios de la máquina de pruebas se usó la tecnología de virtualización *docker*. Éste sistema permite un aislamiento prácticamente completo de la ejecución de ciertos procesos del sistema reutilizando gran parte del sistema operativo del *host*.

Para poder acceder a las tarjetas gráficas de la máquina y aprovechar su rendimiento para el entrenamiento se usó un módulo que trabaja por encima de *docker* llamado *nvidia-docker*, capaz de usar los drivers de las tarjetas instalados en el sistema operativo dentro del sistema virtualizado también.

Complementando a estos, se ha usado *docker-compose* y *nvidia-docker-compose* para facilitar las tareas de despliegue de los contenedores virtualizados y agilizar las pruebas.

Por último, el sistema de *notebooks* de *python*, *jupyter*, se usó como interfaz de pruebas, de control en algunos casos y como interfaz de muestra de resultados.

3

Diseño y desarrollo

3.1. Metodología de trabajo

Los modelos usados para este trabajo son computacionalmente complejos y pueden llegar a tardar varios días en entrenarse. El preprocesado de los datos de entrenamiento tampoco es una tarea pequeña, sobre todo en cuanto a la memoria utilizada. Para evitar perder tiempo de computación recalculando resultados ya obtenidos anteriormente, y para organizar el desarrollo, se decidió construir un marco de trabajo especializado para esta tarea y enfocado al guardado de resultados intermedios, así como a la automatización de la prueba de modelos.

Vamos a ver como funciona observando primero su interfaz de alto nivel y después la mayoría de los módulos desarrollados. Junto con cada descripción aparecerá la interfaz de uso para una mayor claridad en la explicación.

3.1.1. Definición de experimento

Como veremos posteriormente, los datos utilizados para entrenar y probar los modelos debían pasar varias etapas de procesado parametrizables hasta llegar a un formato adecuado para el modelo de aprendizaje automático. Este preprocesamiento se puede descomponer en distintas taréas, con dependencias que se pueden representar en forma de grafo acíclico. Se decidió intentar construir un *framework* especializado para este tipo de tareas que agilizara el trabajo a la hora de probar modelos distintos. Para ello se definieron unas clases de python básicas para organizar la estructura del código. El resultado final son archivos "experimento", parametrizables también, en los que se definen las conexiones entre los distintos pasos de procesamiento y el modelo que se va a usar.

A continuación se muestra un ejemplo en código de como se definiría uno de los modelos probados:

```
def experiment(maxfiles, maxsequence, remove_repeated,
               lstm_size, dropout, batch_size,
               epochs):

    return Sequential([
        XmlLoader(max_files=maxfiles, only_claims=True),
        ClassificationUnwrapper(),
        ColumnFilter(['text', 'section']),
        Parallel(
            lambda input: [input['text'], input['section']],
            [
                Sequential([
                    StopWordRemover(),
                    TextToSequence(),
                    CleanTextSequences(
                        remove_repeated
                    ),
                    Word2Vec(),
                    ChopPadSequences(
                        maxsequence,
                        default_value=np.array([0] * 300)
                    )
                ]),
                ClassificationEncoder('section')
            ],
            lambda results: tuple(results)
        ),
        WeightNormalizer(maxfiles),
        Multiclass(
            maxsequence=maxsequence,
            lstm_size=lstm_size,
            dropout=dropout,
            vector_size=300,
            batch_size=batch_size,
            epochs=epochs
        )
    ])
```

Como se puede ver, la estructura de grafo se consigue gracias a las clases especiales `Sequential` y `Parallel`, que analizaremos en la próxima sección. Estas definiciones de experimentos son muy útiles para después utilizar un pequeño script desde la linea de comandos que lance experimentos. Este script recibe el nombre de archivo de experimento a cargar y los parámetros, y en apenas un par de comandos es capaz de empezar a entrenar el modelo.

Ésto fue especialmente útil al conseguir separar los recursos de la máquina de entrenamiento en cuatro contenedores Docker (con acceso a una tarjeta gráfica cada uno). Se podía lanzar el procesamiento de datos en uno de ellos y cuando éste hubiera empezado a entrenar, lanzar versiones distintas del modelo en el resto de contenedores, aprovechando que los datos procesados ya estaban disponibles en cache.

3.2. Clases base

3.2.1. Step

`Step(id)`

Esta es la clase base de la que heredan todos los distintos pasos de procesamiento de la aplicación, desde la carga de datos a los pasos compuestos como `Sequential` y `Parallel`. Implementa todos los métodos por defecto de un paso, a excepción, de la ejecución en sí. Esto permite definir los pasos más simples como simplemente una clase que hereda de `Step` que implementa el método `all(input, ...)`. Aquellas que requieran diferenciar entrenamiento de predicción pueden implementar solo `train(input, ...)` y `predict(input, ...)`. Además es la encargada de que los resultados del paso se guarden en caché si es necesario, y también el estado del paso si fuera relevante.

3.2.2. ID

`ID([params])`

Cada paso está identificado de manera única con una ID consistente en el nombre de su clase y los parámetros relevantes. Estas IDs permiten composición para construir IDs que no representen solo a un paso sino a ese paso junto con sus dependencias. Esto es especialmente útil para guardar resultados en caché, ya que el resultado al que haya llegado a un paso depende del procesado anterior. Su última responsabilidad es saber generar nombres de archivo válidos y únicos en base a sí misma, especialmente útil a la hora de mantener la caché en el sistema de archivos de la máquina de entrenamiento. Este paso se realizó mediante una función de *hashing* que utiliza el nombre de la clase, los parámetros de ese paso, y las ids de los pasos anteriores como entrada.

3.2.3. Cache

`Cache(type)`

La caché se implementó como una factoría de *singletons* que generaba un objeto global por cada tipo de caché usada. Los distintos tipos de caché usados en la aplicación son *train*, *predict* y *state*. Los primeros dos tipos suelen contener los mismos archivos representando las mismas transformaciones a excepción de los datos iniciales usados. El tipo de cache *state* se ha utilizado para guardar el estado de aquellos módulos que lo requieran, como pueden ser los modelos, pero también los diccionarios tokenizadores. Esta caché se genera durante el entrenamiento y se carga durante la predicción.

La caché se hizo customizable para permitir diversos lugares de guardado de datos, aunque solo se implementó el almacenamiento local a disco duro. Para entrenamiento en máquinas en la nube se podría implementar el guardado a un almacenamiento en red compartido como S3 (Amazon).

3.2.4. Sequential

```
Sequential([steps])
```

Esta clase representa una secuencia de pasos, conectados cada uno con el anterior en los que existe una dependencia directa. Cuando este paso se ejecuta, se comprueba hasta dónde ha sido cacheado el resultado de los subpasos y se continúa desde ese punto, usando como entrada de cada paso la salida del anterior.

3.2.5. Parallel

```
Parallel(split_function, [steps], join_function)
```

Esta clase, aparte de una lista de pasos, recibe dos funciones. Una sirve para separar la entrada en n entradas (iguales o no) para cada subpaso y la otra para juntar los resultados de cada subpaso en uno solo que propagar como resultado general. Los pasos no se ejecutan en paralelo para evitar disputas por los recursos del sistema, pero si que son cacheados de manera independiente, ya que no tienen ninguna dependencia entre ellos.

3.3. Modelos

3.3.1. Multiclass

```
Multiclass(maxsequence, lstm_size, dropout, vector_size, batch_size, epochs)
```

Este modelo representa uno de los más básicos que se puede hacer para clasificación de textos con *lstm*. Es un modelo que consta con dos capas *lstm*, ambas con cantidad de neuronas igual a `lstm_size`. A cada capa se le aplica un dropout igual a `dropout` que aleatoriamente activa y desactiva algunas de sus entradas en cada lote. Además se configura para que la entrada de la red sean sequencias de `maxsequence` vectores de dimensión `vector_size`. Por último tenemos los parámetros `batch_size` y `epochs` para configurar el entrenamiento.

3.3.2. EmbeddingMulticlass

```
EmbeddingMulticlass(maxwords, maxsequence, lstm_size, dropout, vector_size,
                    batch_size, epochs)
```

Este modelo funciona de manera similar a `Multiclass` a excepto por que no espera que la entrada sean secuencias de vectores sino secuencias de enteros (siendo el número más alto igual a `maxwords`). Para realizar la conversión a vector se coloca una capa de mapeo por delante de las capas *lstm*, que se entrena a la vez que el resto de la red para generar vectores relevantes para que las siguientes capas puedan realizar una correcta clasificación.

3.4. Tratamiento de datos

Se han desarrollado muchos pasos de tratamientos de datos útiles para el problema en cuestión pero vamos a repasar solo las más importantes y generalizables a otros sets de datos.

3.4.1. StopWordRemover

```
StopWordRemover()
```

Este módulo sin parámetros se encarga de quitar las palabras vacías, aquellas sin significado como artículos, pronombres, preposiciones, etc... Éstas, conocidas como *stop words* en inglés, se consideran poco útiles para el análisis de lenguaje natural para algunos problemas.

3.4.2. TextToSequence

```
TextToSequence()
```

Módulo de procesado de texto que lo convierte en una lista de palabras, quitando los caracteres especiales (puntos, comas, etc...).

3.4.3. CleanTextSequences

```
CleanTextSequences(remove_repeated)
```

Módulo de limpieza de secuencias de texto como las generadas por el módulo anterior que realiza un análisis de frecuencia de las palabras de todos los textos para decidir qué palabras no son relevantes para la clasificación. Los criterios se escogieron tras un análisis manual de los datos de las patentes. En concreto, quita de cada secuencia todas las palabras que son únicas en todos los textos (ya que, en principio, el modelo no podría generalizar nada con ellas y no servirían para clasificar patentes nunca vistas). Además quita las palabras que aparezcan en más de un 90 % de las patentes, considerandolas palabras vacías por ser tan comunes. Opcionalmente, puede quitar las ocurrencias repetidas de una palabra dentro de una sola patente.

La lista de palabras a eliminar se obtiene solo de los datos de entrenamiento y se guarda en el estado del módulo para usar en predicción. De esta manera se evita que esta selección esté contaminada por los datos de test.

Como se verá posterioremente en el análisis, todas estas operaciones reducen considerablemente la longitud de los textos de las patentes haciéndo más fácil y rápido su uso.

3.4.4. Word2Vec

```
Word2Vec()
```

Este paso de transformación convierte cada palabra en un vector de 300 dimensiones, basándose en un corpus ya entrenado por Google con noticias. Si una palabra no existe dentro del corpus, es ignorada.

3.4.5. TextToIndexSequence

```
TextToIndexSequence(maxwords)
```

A pesar de ser claramente similar a `TextToSequence`, este módulo, a parte de dividir en palabras, codifica cada una como un entero único, hasta `maxwords` palabras (escogiendo las más usadas).

Los datos que usa para establecer esta correspondencia entre palabras y enteros son los de entrenamiento, por lo que este módulo tiene como estado ese diccionario de conversión que usa posteriormente sobre los datos a predecir. Esto evita que los datos de test contaminen los datos de entrenamiento y puedan salir resultados sesgados.

3.4.6. ChopPadSequences

```
ChopPadSequences(maxsequence, default_value)
```

Como hemos visto antes, a pesar de tener neuronas *lstm* que pueden procesar texto de longitud variable, el algoritmo de aprendizaje funciona mucho mejor si las secuencias usadas para el entrenamiento tienen la misma longitud. Por eso la función de esta clase es recortar las secuencias más largas que un valor dado (`maxsequence`) y ampliar aquellas que sean más cortas añadiendo valores nulos al principio. Este valor nulo es un 0 en el caso de secuencias de índices que representan palabras y un vector $\bar{0}$ en el caso de las secuencias de vectores, para eso está el parámetro `default_value`.

3.4.7. WeightNormalizer

```
WeightNormalizer(max_samples)
```

Las categorías de la mayoría de niveles de las patentes no están equilibradas. Por ejemplo, en el primer nivel un 70 % de las patentes están clasificadas en la clase G, seguidas de un 38 % de clase H. Las siguientes clases no llegan al representar el 1 % de las patentes. Como nuestros modelos ni siquiera intentarían predecir clases tan minoritarias por suponer tan poco coste podemos usar este módulo que cambia las proporciones entre clases para hacerlas más representativas.

Desgraciadamente no es del todo fácil de realizar ya que como este problema es multietiqueta, resulta que tan solo 14 patentes de las 100592 no pertenecen a ninguna de las dos clases mayoritarias. Esto causa que una proporcionalidad perfecta sea imposible de alcanzar. Aun así, este módulo garantiza un mínimo de ocurrencias (`max_samples` / #categorías) de cada categoría, devolviendo hasta `max_samples` filas.

Para conseguir garantizar un mínimo de ocurrencias se repiten datos de las clases minoritarias y se quitan datos de las clases mayoritarias. Aun así, debido a que el problema es multietiqueta, las proporciones entre clases que salen de este módulo no son equilibradas, pero si mejores que sin el tratamiento.

3.4.8. ClassificationEncoder

`ClassificationEncoder(target)`

Las categorías vienen expresadas como una lista de números, pero la salida de nuestros modelos tiene que ser un vector, así que este módulo se encarga de realizar al conversión. Está conversión es comunmente conocida como *k-hot-encoding*, una variante de *one-hot-encoding*. Este algoritmo crea n columnas, cada una representando una categoría y coloca 1 en las clasificaciones que le corresponden a cada patente y 0 en el resto.

3.5. Metodología de entrenamiento

Una buena metodología de entrenamiento es esencial para conseguir modelos que adquieran buena precisión y, sobretodo, que sean útiles con sus predicciones. En este trabajo se ha tenido especial atención en la división de los datos disponibles y en el cálculo de la función de optimización del modelo.

3.5.1. Dividisión entre entrenamiento, test y validación

Primero veamos por qué es importante realizar esta división en tres categorías. Los datos de entrenamiento son los que vamos a utilizar para entrenar tanto el modelo como las capas de procesamiento de datos que tengan estado. El set de datos de validación se usará para comprobar cómo funciona el modelo con datos no observados anteriormente. Si repetimos esto con cada modelo probado se obtendrá una métrica de precisión para cada uno, y en base a ella podemos escoger qué modelo es el definitivo. Pero claro, al haber escogido entre modelos usando esa precisión, nuestra decisión está sesgada y para obtener una estimación del rendimiento real debemos usar el set de datos de test, que no ha sido usado anteriormente.

Los datos utilizados para este proyecto han sido las patentes presentadas a la Oficina de Patentes y Marcas de Estados unidos entre los años 2013 y 2015. Para 2013 y 2014 se disponían de todos los datos mientras que de 2015 solo los primeros meses.

Una opción disponible es coger todas las patentes y aleatoriamente repartirlas en los tres sets, pero dado el problema subyacente y que se quiere valorar la utilidad del modelo, se cogeran las patentes teniendo en cuenta el factor temporal. El test de entrenamiento del tamaño que se requiera, se sacará de las patentes de 2013, el set de validación de las patentes de 2014 y el set de test serán todas las patentes disponibles de 2015. Esta división es útil para valorar el modelo puesto que se quiere usar para patentes nuevas no vistas anteriormente, teniendo en cuenta todo el histórico de clasificación.

3.5.2. Optimización del modelo

Como hemos visto, los modelos de aprendizaje automático intentan minimizar una función calculando su gradiente y actualizando los pesos de las neuronas de manera que su resultado se aproxime cada vez más a la función objetivo. La cantidad por la que se actualizan los pesos de esta función viene dada por el optimizador usado.

Para estos experimentos la función de pérdida es la entropía cruzada binaria. Esta queda definida por la fórmula:

$$\text{Perdida}(y, \hat{y}) = -\frac{1}{N}\sum_{i}^{N} [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)] \tag{3.1}$$

En donde y es la clasificación correcta, $\hat{y}$ la que se ha predicho y N es el número de categorías.

Esta función es un caso particular de la entropía cruzada, para variables discretas binarias. Mide cómo de diferentes son las distribuciones de y y $\hat{y}$, de manera que si no son distinguibles devolverá 0.

$$\text{Entropía cruzada discreta}(p, q) = -\sum_{x} p(x) \log q(x) \tag{3.2}$$

Para deducir la función final que se ha usado se puede tener en cuenta que p y q son variables binarias (o pertenece a una clase o no pertenece) y que se pueden modelar como distribuciones de Bernoulli de parámetros y e $\hat{y}$ respectivamente. Ésto nos deja que las probabilidades de cada una para 0 y 1 son $p = \{y, 1 - y\}$ y $q = \{\hat{y}, 1 - \hat{y}\}$. Sustituyendo en la fórmula (3.2) tenemos:

$$-\sum_{x} p(x) \log q(x) \underset{x \in \{0,1\}}{=} -[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})] \tag{3.3}$$

Sumando por cada una de las clases obtenemos la formula (3.1). Queda sumar por todos los casos de ejemplo en el set de datos y el valor resultante indicará lo cerca que está nuestro modelo de predecir correctamente todos los ejemplos. Este valor es el que intenta minimizar el optimizador.

El optimizador usado durante este trabajo es Adam, un método optimización basado en *Stochastic Gradient Descent* considerado estado del arte en optimización de reder neuronales. Los parámetros usados fueron los parámetros por defecto descritos en **??**.

4

Pruebas y Resultados

En este capítulo se repasarán las pruebas hechas usando el *framework* contado en el capítulo anterior. Cómo se han ido escogiendo los modelos, sus parámetros, los parámetros del procesamiento de datos, etc... Además se compararán los resultados con algunos de los resultados relatados en el capítulo 2 para concluir que hace falta más trabajo, pruebas y tiempo de entrenamiento para encontrar modelos que funcionen tan bien como los vistos anteriormente.

Aun así, también se ha realizado un análisis de los datos para fundamentar las decisiones en cuanto a ciertos parámetros, cuyos resultados serán descritos aquí.

Por motivos de rendimiento, se tuvo que reducir la cantidad de texto a analizar para que fuera posible trabajar con ellos en el tiempo disponible. Por esta razón se ha decidido tomar de cada patente solo el texto extraído de las reivindicaciones, que se estima que será lo suficientemente representativo a la vez que escueto. Tanto el entrenamiento como el análisis preliminar de los datos fue realizado tomando solo las reivindicaciones.

4.1. Metodología

Con tantos parámetros a la hora de escoger modelo, se ha empezado con un modelo simple escogido arbitrariamente para comprobar la efectividad de la función de pérdida y algoritmo de optimización elegidos. También se comprobó que ese método fuera capaz de sobreajustar un conjunto de datos pequeño. A continuación, observando los resultados obtenidos en cada prueba se han ido realizando alteraciones, algunas más pequeñas y otras más drásticas de los parámetros y estructura del modelo; basándose en una métrica de rendimiento.

4.1.1. Métrica de rendimiento

Para evaluar el rendimiento de cada modelo se ha usado la medida-F (conocida también como F1 score). Es una medida que considera tanto la precisión como la exhaustividad.

La precisión la definimos como el número de etiquetas predichas que son correctas:

$$\text{precision(x)} = \frac{|\{\text{patentes con categoría x}\} \cap \{\text{patentes para las que se predice x}\}|}{|\{\text{patentes para las que se predice x}\}|} \tag{4.1}$$

Por otro lado, la exhaustividad mide cuantas de las patentes de una categoría son identificadas correctamente:

$$\text{exhaustividad(x)} = \frac{|\{\text{patentes con categoría x}\} \cap \{\text{patentes para las que se predice x}\}|}{|\{\text{patentes con categoría x}\}|} \quad (4.2)$$

El objetivo es conseguir que ambas se aproximen lo máximo posible a 1. Para medir como de cerca están vamos a usar la medida-F, que no es más que la media armónica de la precisión y la exhaustividad:

$$F_1 = 2\frac{\text{precisión} \cdot \text{exhaustividad}}{\text{precisión} + \text{exhaustividad}} \quad (4.3)$$

Para conseguir la medida-F para todas las clases se realiza una media ponderada de las distintas medidas-F usando la proporción de cada una de las clases.

Para agilizar las pruebas, se programó cada modelo para que evaluara los datos de validación a la finalización de cada época, para ir obteniendo medidas del avance del entrenamiento de los modelos en directo.

4.2. Análisis del corpus

Con la ayuda de un *notebook* de *Jupyter* se evaluaron los datos de entrenamiento para intentar medir la información que se perdería con cada transformación de los datos. Para estos análisis se cogieron los datos de entrenamiento (patentes de 2013) para evitar sesgar las decisiones a tomar.

Para muchas métricas se observa el percentil 90 de la distribución para poder idear transformaciones que no se vean condicionadas por valores atípicos y que sirvan para una gran mayoría de los datos de entrada.

4.2.1. Distribución de clases

Empezamos midiendo los datos sobre las clases a las que pertenece cada patente. En total hay 100592 patentes y 8 categorías. Como ya se ha visto, la frecuencia de las distintas clases no está nada equilibrada (figura 4.1 y Distribución de las clases en los datos de entrenamiento))

	A	B	C	D	E	F	G	H
# de patentes	1169	1630	1047	31	178	537	70203	28462
% de patentes	1.16 %	1.62 %	1 %	0.03 %	0.18 %	0.53 %	69.79 %	38.23 %

Tabla 4.1: Distribución de las clases en los datos de entrenamiento

Como se ha comentado anteriormente, el número de patentes que no pertenecen a la categoría G o H es 14.

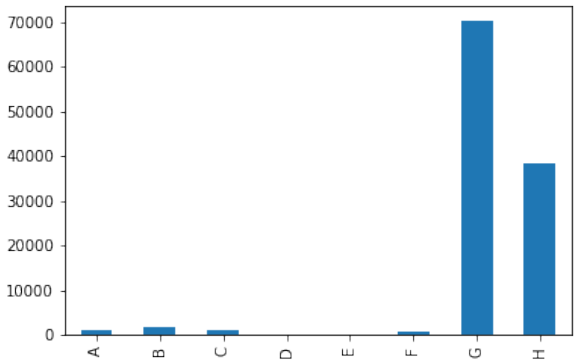

Figura 4.1: Distribución de las clases en los datos de entrenamiento

4.2.2. Frecuencia de palabras

Sin ningún tipo de procesado (a excepción de la extracción de palabras del texto) obtenemos la distribución de frecuencias de las palabras mostrada en la figura 4.2 y en la siguiente lista de palabras más frecuentes: the, of, a, to, and, wherein, in, is, claim, first, for, one, said, second, an, data, at, method, by, device.

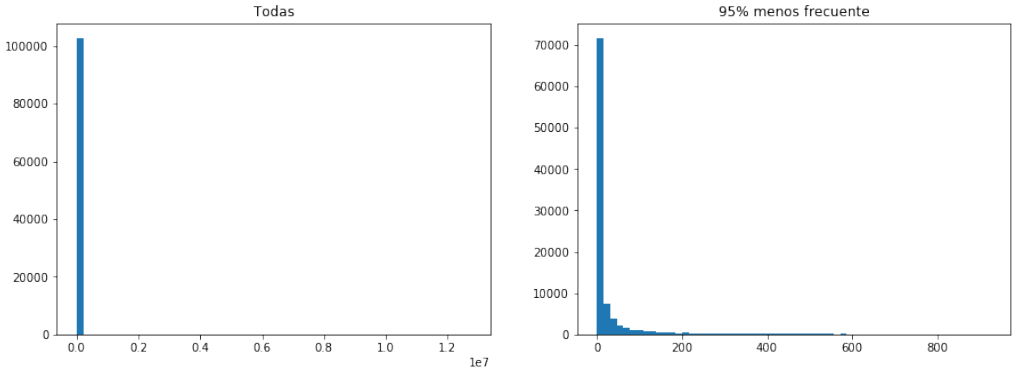

Figura 4.2: Cantidad de palabras (y) que tiene una frecuencia (x)

Como se puede observar claramente, hay muchas palabras muy poco frecuentes y el resto no llegan a apreciarse en la gráfica. Se puede intentar mostrar solo el 95 % menos frecuente, pero vemos que la distribución sigue siendo similar.

La lista de palabras más frecuentes nos lleva a pensar que quitar las palabras vacías probablemente sea una buena idea. Una vez realizada esta tarea, la gráfica anterior sigue teniendo la misma pinta (figura 4.3), pero la lista de palabras más frecuentes nos da otro tipo de resultados: wherein, claim, first, one, said, second, data, method, device, 1, comprising, least, information, system, plurality, image, signal, computer, comprises, user.

Frecuencia de palabras en patentes

Pasamos a analizar el uso que tiene cada palabra respecto al número de patentes en el que aparece. Tomando el set de datos sin palabras vacías vemos que la cantidad de patentes en las que aparece cada palabra sigue una distribución similar al análisis anterior: Hay muchas palabras que aparecen en muy pocas patentes y muy pocas palabras que aparezcan en muchas patentes. Por esto se decidió quitar, al igual que con las palabras vacías, las palabras que aparecieran en más del 90 % de las patentes o las que aparecieran solo una vez, asumiendo que no tienen mucho poder para inferir buenas generalizaciones. El número de palabras diferentes que aparece en todos los textos tras estos filtros es: 45695.

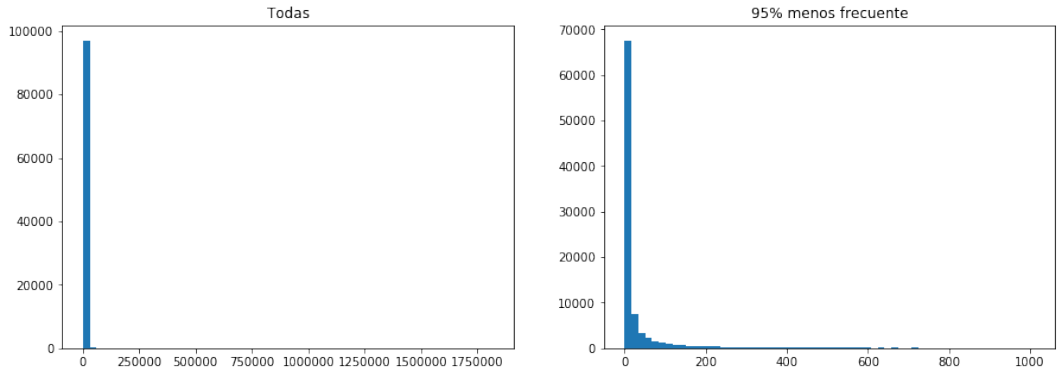

Figura 4.3: Cantidad de palabras (y) que tiene una frecuencia (x) (sin palabras vacías)

4.2.3. Longitud del texto

Por último, se midió la longitud del texto para cada patente al pasar distintos filtros. Todos los resultados a continuación fueron obtenidos a partir de las reivindicaciones. Los gráficos muestran el número de patentes (y) que tiene una longitud concreta (x) aunque, en ánimo de mejorar los gráficos, están recortados en el percentil 90.

Texto sin filtrar

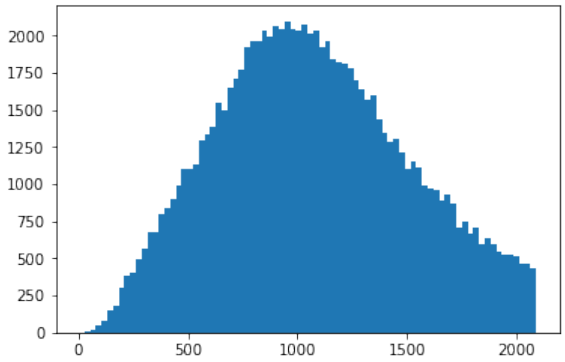

Figura 4.4: Distribución de las longitudes de textos (Sin tratar)

Longitud mínima: 25

Longitud máxima: 23013

Percentil 90: 2091

Percentil 80: 1668

Percentil 70: 1427

Texto sin palabras vacías

Claramente hay muchas palabras vacías dentro de las secuencias. El percentil 90 se ve reducido prácticamente a la mitad.

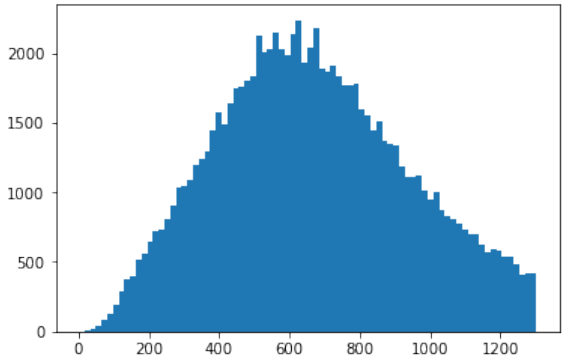

Figura 4.5: Distribución de las longitudes de textos (Quitando palabras vacías)

Longitud mínima: 15

Longitud máxima: 22573

Percentil 90: 1303

Percentil 80: 1036

Percentil 70: 887

Texto sin palabras vacías, palabras únicas y palabras demasiado frecuentes

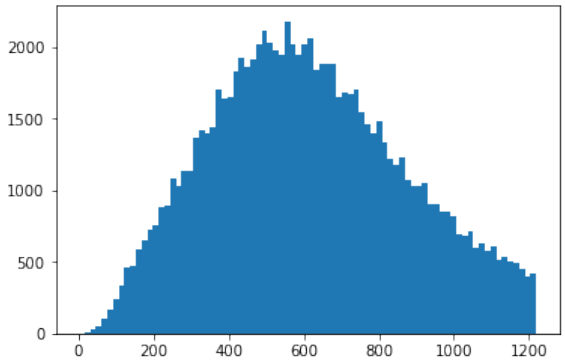

Figura 4.6: Distribución de las longitudes de textos (Quitando palabras vacías e implementando la limpieza descrita)

Longitud mínima: 12

Longitud máxima: 21606

Percentil 90: 1220

Percentil 80: 965

Percentil 70: 822

Este cambio no tiene un efecto tan significativo como el anterior. aunque no es despreciable.

Texto sin palabras vacías, palabras únicas y palabras demasiado frecuentes y sin palabras repetidas

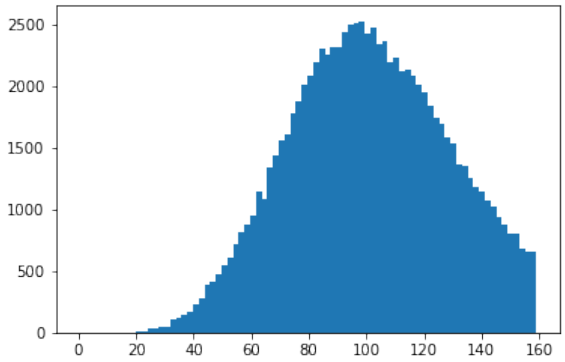

Figura 4.7: Distribución de las longitudes de textos (Quitando las palabras vacías, con limpieza y quitando duplicados)

Longitud mínima: 9

Longitud máxima: 1592

Percentil 90: 159

Percentil 80: 137

Percentil 70: 124

Al parecer las palabras dentro de estos textos se repiten mucho. Esta última transformación consiste en solo dejar dentro de una secuencia la primera ocurrencia de cada palabra. Como se puede observar, puede ser muy útil si necesitamos reducir la longitud de las secuencias de entrenamiento, ya consigue un factor de reducción del 80 %. Sin embargo es posible que esta alteración sea demasiada y tenga un impacto muy negativo en la capacidad predictiva del modelo. Precisamente se van a usar redes *lstm* en el modelo por su capacidad para recibir secuencias como entrada, pero si se quitan tantas palabras habrá "frases" enteras que se pierdan, perdiendo no solo las palabras individualmente si no su significado colectivo como frase.

4.3. Resultados

Se han evaluado bastantes modelos y combinaciones de parámetros, pero aquí solo se van a mostrar los más relevantes. Para que nos resulte más fácil el identificar a cada uno, se les asignaran letras del alfabeto latino. Además, para no repetirlo para cada modelo queda especificado aquí que todos los modelos acaban con una capa de perceptrones de función de activación sigmoidal y de tamaño igual al número de categorías.

El mayor fallo que tienen todos estos modelos es el haber sido entrenados con un conjunto de datos excesivamente pequeño (5-10 % de las patentes de un año), por problemas con el procesado de datos y la gestión de las secuencias en memoria. Es probable que alguno de estos modelos funcione mucho mejor de lo obtenido aquí con una cantidad de datos más grande, pero esto se deja como posible ampliación del trabajo.

Todas las tablas de métricas han sido obtenidas usando datos de validación, diferentes de los de test.

Modelo A

La primera prueba realizada fue tomando prámetros bastante arbitrarios: las primeras 1200 palabras de una patente, con un vocabulario de tamaño 20000 (considerablemente menos que el número de palabras diferentes) y un modelo con un mapeo y dos capas *lstm* de 150 neuronas cada una y un *dropout* entre capas de 0,2. El tamaño de lote que se uso fue bastante pequeño (15), para 5000 patentes.

Todos los pesos, tanto los del diccionario de mapeo como los de las neuronas, fueron iniciados aleatoriamente con el incializador aleatorio Glorot.

La métrica para cada clase habiendo pasado 30 épocas la podemos observar en la tabla 4.2.

metric	A	B	C	D	E	F	G	H	global
precision	0	0	0	0	0	0	0.78	0.58	0.72
recall	0	0	0	0	0	0	0.89	0.65	0.76
F1	0	0	0	0	0	0	0.83	0.61	0.74

Tabla 4.2: Modelo A tras 30 épocas

Pasadas más épocas podemos ver como, a costa de perder precisión en las clases mayoritarias, el modelo ha conseguido algo más de variedad. La tabla 4.3 nos muestra los resultados.

metric	A	B	C	D	E	F	G	H	global
precision	0.04	0	0.12	0	0	0	0.74	0.41	0.58
recall	0.04	0	0.08	0	0	0	0.80	0.52	0.66
F1	0.04	0	0.10	0	0	0	0.77	0.46	0.62

Tabla 4.3: Modelo A tras 50 épocas

Por desgracia, este modelo no consiguó acercarse a nada más en las siguientes épocas. Además, la función de pérdida sobre el conjunto de datos de entrenamiento ya había convergido a 0,01, mientras que la precisión había subido hasta 0,99. Claramente este modelo era capaz de sobreajustar los datos de entrada.

Modelo B

Pensando que el modelo anterior no iba mal encaminado, que quizás sufrió sobreajuste por tener pocos datos, probamos a realizar el mismo modelo, pero entrenado en GPU con lotes de 100 patentes de un total de 80000

metric	A	B	C	D	E	F	G	H	global
precision	0.29	0.21	0.45	0	0.09	0.13	0.86	0.66	0.75
recall	0.17	0.20	0.46	0	0.09	0.17	0.81	0.78	0.76
F1	0.21	0.20	0.46	0	0.08	0.14	0.84	0.71	0.75

Tabla 4.4: Modelo B tras 45 épocas

Al igual que en el modelo anterior, el aprendizaje fue interrumpido por un estancamiento de la función de pérdida. Además, este análisis tardó aproximadamente 6000 segundos en completar cada época. Haciéndo los cálculos, este modelo estuvo entrenando unas 75 horas.

Modelo C

En este momento se decide probar los métodos sin entrenado de mapeo para ver si así se reduce el tiempo de entrenamiento al reducir considerablemente el número de pesos del modelo y teniendo un mapeo ya realizado.

metric	A	B	C	D	E	F	G	H	global
precision	0.27	0.17	0.39	0	0	0.08	0.86	0.66	0.71
recall	0.20	0.42	0.77	0	0	0.15	0.85	0.79	0.79
F1	0.23	0.25	0.52	0	0	0.11	0.86	0.72	0.75

Tabla 4.5: Modelo C tras 85 épocas

Desgraciadamente el modelo no parece mucho mejor, de hecho, hasta deja de clasificar la clase E que si había conseguido el modelo anterior. Así que se decide a partir de este modelo cambiar varios parámetros de manera independiente para intentar entender el efecto que tienen sobre el modelo.

Modelo D

Se probó a aumentar el dropout para quizás evitar el sobreajuste y conseguir una mejor generalización, pero su mayor efecto fue la ralentización del entrenamiento. Es posible que continuando el entrenamiento se llegue a resultados parecidos a los de modelos anteriores, pero no merece la pena.

metric	A	B	C	D	E	F	G	H	global
precision	0.12	0	0.41	0	0	0.08	0.72	0.53	0.63
recall	0.10	0	0.83	0	0	0.04	0.91	0.71	0.79
F1	0.11	0	0.56	0	0	0.05	0.80	0.61	0.70

Tabla 4.6: Modelo D tras 100 épocas

Modelo E

Para continuar se probó a variar el número de neuronas en cada capa, probando primero a aumentarlas. Los resultados vuelven a ser parecidos a los obtenidos por otros modelos aunque, como en el caso anterior, el entrenamiento se volvió más lento y hubo que bajar el tamaño de lote por no caber tantos parámetros en la GPU.

metric	A	B	C	D	E	F	G	H	global
precision	0.4	0.26	0.46	0	0	0.27	0.84	0.71	0.76
recall	0.20	0.30	0.63	0	0	0.11	0.85	0.69	0.76
F1	0.27	0.28	0.53	0	0	0.16	0.85	0.70	0.76

Tabla 4.7: Modelo E tras 100 épocas

Modelo F

Intentando conseguir iteraciones más rápidas en la prueba de modelos se probó a cortar agresivamente el tamaño de las entradas y un poco de la red neuronal para ver que tipo de resultados se obtenían. Este modelo utilizó secuencias de 150 palabras obtenidas primero quitando las palabras repetidas y después truncando o haciendo padding hasta 150. A pesar de no ser un gran acierto para redes *lstm*, que podrían aprender el orden de las palabras y cierta estructura que estamos eliminando, los resultados mostraron que no se había perdido mucha información, aunque cabe decir que se aumentó el número de datos de entrenamiento al doble.

metric	A	B	C	D	E	F	G	H	global
precision	0.16	0.15	0.39	0	0.13	0.11	0.89	0.67	0.73
recall	0.24	0.32	0.61	0	0.15	0.22	0.83	0.78	0.78
F1	0.19	0.21	0.47	0	0.14	0.15	0.86	0.72	0.75

Tabla 4.8: Modelo F tras 100 épocas

Modelo G

Estudiando como bajaba la pérdida de entrenamiento y no la de validación surgió la hipótesis de que los modelos estaban tendiendo al sobreajuste, así que se decidió bajar el número de neuronas por capa hasta 64 manteniendo el resto de parámetros. Aunque los resultados parezcan similares a los de otros modelos, durante el entrenamiento se observó como la pérdida de entrenamiento y la pérdida de validación, aun distantes, bajaron de manera más sincronizada que en los modelos anteriores.

metric	A	B	C	D	E	F	G	H	global
precision	0.13	0.13	0.42	0	0	0.16	0.85	0.9	0.70
recall	0.33	0.47	0.68	0	0	0.14	0.90	0.70	0.80
F1	0.18	0.21	0.52	0	0	0.15	0.87	0.69	0.75

Tabla 4.9: Modelo G tras 100 épocas

Modelo H

El último modelo que se probó consistió en una reducción todavía mayor de las neuronas en la capa de la red (hasta 32). Sorprendentemente, este modelo fue capaz de adquirir una capacidad predictiva similar a los mejores modelos anteriores, eso si, necesitó más épocas para aprender. Este modelo, por su velocidad al constar de muy pocas neuronas, fue entrenado hasta un número muy alto de épocas para comprobar si avanzaba y conseguía predecir mejor, pero ya se había quedado estancado.

metric	A	B	C	D	E	F	G	H	global
precision	0.18	0.14	0.43	0	0.20	0.09	0.89	0.66	0.71
recall	0.34	0.38	0.63	0	0.08	0.18	0.78	0.79	0.76
F1	0.23	0.20	0.50	0	0.11	0.12	0.83	0.72	0.74

Tabla 4.10: Modelo H tras 400 épocas

Los mejores modelos que hemos visto han obtenido todos los mismos resultados, lo que nos lleva a pensar que esa es prácticamente toda la información que podemos recuperar de los datos de entrenamiento usados.

5

Conclusiones y trabajo futuro

5.1. Conclusiones

Este trabajo se ha desarrollado pensando en familiarizarse con las redes *lstm* y los problemas de clasificación de texto. El capítulo Estado del Arte ha supuesto un análisis del funcionamiento de muchos modelos de aprendizaje automático así como del funcionamiento de las redes *lstm* en concreto. Tanto su estructura general como su motivación o cómo funciona su entrenamiento. Además se han explorado otros conceptos básicos de la ciencia de datos y, en concreto, el procesamiento de texto para modelos automáticos.

La implementación de un *framework* específico para las pruebas en este proyecto supone una gran ayuda para futuras investigaciones y la continuación de este proyecto. No hubiera sido posible probar tantos modelos con variaciones de los mismos datos tan rapidamente de no haber sido por esa infraestructura. Sin mencionar la simplicidad que aporta a los módulos de tratamiento de datos y aprendizaje automático.

No se ha llegado a ningún modelo a la altura del estado del arte de la clasificación de patentes, pero con las pruebas realizadas se ha obtenido un camino a seguir e ideas para llegar a conseguir entrenar un modelo con neuronas *lstm* para esta tarea de clasificación.

5.2. Trabajo futuro

Claramente se puede seguir investigando este método de clasificación de texto con la meta de conseguir un modelo con buena precisión. Pero también se han visto otras alternativas en los artículos citados sobre clasificación de patentes. Por ejemplo, sería interesante probar uno de los modelos con mapeo probados pero introduciendo una etapa de *stemming* delante para no evaluar palabras sino lexemas. Por la naturaleza del lenguaje científico de las patentes, es plausible pensar que puede ser de gran ayuda. Por supuesto, también se puede intentar implementar modelos totalmente distintos sin *lstm* como el descrito en el capítulo 2, en el que crean una firma por cada patente.

Por último, la infraestructura desarrollada para realizar los tests tiene hueco para mejora y ampliación que podría ser muy útil en caso de seguir probando modelos. Una importante mejora será la capacidad de trabajar en batches, u offline para poder procesar grandes cantidades de datos incluso con escasez de memoria RAM.

Bibliografía

[1] Long Short-Term Memory | Neural Computation | MIT Press Journals.

[2] Understanding LSTM Networks – colah's blog.

[3] Latent semantic analysis. *Wikipedia*, June 2017.

[4] Andreas Buja, Werner Stuetzle, and Yi Shen. Loss functions for binary class probability estimation and classification: Structure and applications. *Working draft, November*, 2005.

[5] Ian Christopher, Sydney Lin, and Sigurd Spieckermann. Automated Patent Classification. 2011.

[6] Mark Dredze, Partha Pratim Talukdar, and Koby Crammer. Sequence learning from data with multiple labels. In *Workshop Co-Chairs*, page 39, 2009.

[7] C. J. Fall, K. Benzineb, J. Guyot, A. Törcsvári, and P. Fiévet. Computer-assisted categorization of patent documents in the international patent classification. In *Proceedings of the International Chemical Information Conference (ICIC'03), Nîmes, France*, 2003.

[8] Yarin Gal and Zoubin Ghahramani. A Theoretically Grounded Application of Dropout in Recurrent Neural Networks. *arXiv:1512.05287 [stat]*, December 2015. arXiv: 1512.05287.

[9] Klaus Greff, Rupesh Kumar Srivastava, Jan Koutník, Bas R. Steunebrink, and Jürgen Schmidhuber. LSTM: A Search Space Odyssey. *arXiv:1503.04069 [cs]*, March 2015. arXiv: 1503.04069.

[10] Andrej Karpathy, Justin Johnson, and Li Fei-Fei. Visualizing and understanding recurrent networks. *arXiv preprint arXiv:1506.02078*, 2015.

[11] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv:1412.6980 [cs]*, December 2014. arXiv: 1412.6980.

[12] Michael A. Nielsen. Neural Networks and Deep Learning. 2015.

[13] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

[14] Dilesha Seneviratne, Shlomo Geva, Guido Zuccon, Gabriela Ferraro, Timothy Chappell, and Magali Meireles. A signature approach to patent classification. In *Asia Information Retrieval Symposium*, pages 413–419. Springer, 2015.

[15] Marina Sokolova and Guy Lapalme. A systematic analysis of performance measures for classification tasks. *Information Processing & Management*, 45(4):427–437, July 2009.

[16] Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[17] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[18] Yin Zhang, Rong Jin, and Zhi-Hua Zhou. Understanding bag-of-words model: a statistical framework. *International Journal of Machine Learning and Cybernetics*, 1(1-4):43–52, December 2010.